****МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

**Практикум по курсу**

**"Суперкомпьютеры и параллельная обработка данных"**

**Разработка параллельной версии программы для сортировки данных.**

**Сортировка массива целых чисел при помощи метода быстрой сортировки (Quick Sort).**

**ОТЧЕТ**

**о выполненном задании**

студента 324 учебной группы факультета ВМК МГУ

Короткова Бориса Сергеевича

Москва, 2018 г.

Оглавление

[1 Постановка задачи - 2 -](#_Toc528419256)

[2 Описание алгоритма быстрой сортировки массива целых чисел - 2 -](#_Toc528419257)

[2.1 Основа: последовательный алгоритм - 2 -](#_Toc528419258)

[2.2 Параллельный алгоритм - 3 -](#_Toc528419259)

[3 Результаты замеров времени выполнения - 4 -](#_Toc528419260)

[3.1 Таблицы и графики - 4 -](#_Toc528419261)

[3.1.1 OpenMP на Polus - 4 -](#_Toc528419262)

[3.1.2 OpenMP на Bluegene - 6 -](#_Toc528419263)

[3.1.3 OpenMP на ноутбуке - 7 -](#_Toc528419264)

[4 Анализ результатов - 8 -](#_Toc528419265)

[5 Выводы - 8 -](#_Toc528419266)

# Постановка задачи

Перед нами поставлена задача отсортировать массив целых чисел по не убыванию при помощи алгоритма быстрой сортировки. Результатом выполнения программы является полностью отсортированный массив.

Требуется:

1. Реализовать сортировку массива целых чисел любого размера с помощью технологии параллельного программирования OpenMP.
2. Сравнить её эффективность на массивах разного размера.
3. Исследовать масштабируемость полученных программ и построить графики зависимости времени выполнения программ от числа используемых потоков и объёма входных данных.

# Описание алгоритма быстрой сортировки массива целых чисел

## Основа: последовательный алгоритм

Простейшая форма алгоритма быстрой сортировки массива целых чисел имеет следующий вид:

1. /\*!
2. \* quickSort - реализация алгоритма быстрой сортировки
3. \* @param a сортируемый массив
4. \* @param n индекс последнего элемента массива (не размер массива!)
5. \*/
6. **void** quickSort(**float**\* a, **const** **long** n) {
7. **long** i = 0, j = n;
8. **float** pivot = a[n / 2]; // выбор опорного элемента
9. **do** {
10. while (a[i] < pivot) i++;
11. while (a[j] > pivot) j--;
12. **if** (i <= j) {
13. std::swap(a[i], a[j]);
14. i++; j--;
15. }
16. } **while** (i <= j);
17. **if** (j > 0) quickSort(a, j);
18. if (n > i) quickSort(a + i, n - i);
19. }

* Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем O(n\*log(n)) обменов при упорядочении n элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

**Общая идея алгоритма состоит в следующем:**

* Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.
* Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньшие опорного», «равные» и «большие».
* Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

## Параллельный алгоритм

В алгоритме быстрой сортировки, исходный массив разбивается на 2 части, обработка которых ведется независимо (поэтому может выполняться параллельно). Задачи должны создаваться внутри параллельной области, однако если мы поместим директиву omp parallel внутрь функции, то у нас будут рекурсивно создаваться потоки. Создание потока — очень сложная операция, требующая значительных вычислительных затрат, поэтому потоки мы создадим до вызова функции.

1. #pragma omp parallel shared(a)
2. {
3. #pragma omp single nowait
4. {
5. quickSort(a, n - 1);
6. } // #pragma omp single
7. } // #pragma omp parallel

Функция QuickSort должна быть вызвана только один раз — поэтому вызов помещен в область omp single. И так, поток, выполнивший область omp single добавит в пул одну или две задачи и завершится. Но мы должны дождаться пока эти задачи будут выполнены. При этом, добавлять задачи в пул будет не только этот поток, но любой другой поток (ведь он может взять из пула задачу и обработать часть массива). Каждый поток после порождения своих задач, должен дождаться их завершения.

1. #pragma omp task shared(a)
2. if (j > 0) quickSort(a, j);
3. #pragma omp task shared(a)
4. if (n > i) quickSort(a + i, n - i);
5. #pragma omp taskwait

Также оптимизируем нашу программу. Для этого, при небольшом кол-ве данных будем проводить последовательную сортировку, чтобы не тратить время на создание потоков.

1. if (n < 100) { // если размер массива меньше 100
2. // сортировка выполняется в текущем потоке
3. **if** (j > 0) quickSort(a, j);
4. if (n > i) quickSort(a + i, n - i);
5. **return**;
6. }

# Результаты замеров времени выполнения

Ниже приведены результаты замеров времени программ на суперкомпьютерах Bluegene и Polus: непосредственно в табличной форме и наглядно на 3D-графиках.

Программа была запущена в конфигурациях:

* на Polus - 1,2,4,8,16,32,64,128 потоков для OpenMP-программы;
* на Bluegene, к сожалению, установлен старый IBM`овский компилятор, который не знает #pragma task.

Поэтому мне не удалось запустить мою программу на данной машине. Компилятор g++ же не знает флага –fopenmp, а команда –qsmp=omp также не распознаётся для данного компилятора. (Скрины см.ниже)

Также для сравнения программа была запущена на ноутбуке (Core i7-3630QM 2.40GHz × 4, 8GB RAM, Windows 10)

Каждая конфигурация была запущена 3 раза. Ниже приведены усредненные результаты.

## Таблицы и графики

### OpenMP на Polus

**Компиляция и запуск:**

1. xlc++\_r –qsmp=omp QuickSort.cpp –o sort
2. mpisubmit.pl –n 1 –t <кол-во потоков> ./sort (также нужно в программе изменить параметр *n – кол-во элементов массива* и *flowes – кол-во потоков*)
3. cat sort.<номер>.out (для печати отсортированного массива нужно убрать комментарии в программе)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| #threads/size | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| 1 | 0,00744923 | 0,00747729 | 0,00753891 | 0,0083065 | 0,0161649 | 0,0997805 | 0,948261 |
| 2 | 0,00750996 | 0,00750652 | 0,00761756 | 0,00877215 | 0,0151283 | 0,0980296 | 0,722503 |
| 4 | 0,00755147 | 0,00760204 | 0,00773733 | 0,00825829 | 0,0148293 | 0,0758358 | 0,702355 |
| 8 | 0,00773349 | 0,00775673 | 0,00790431 | 0,00854018 | 0,0140687 | 0,0652352 | 0,572813 |
| 16 | 0,00802679 | 0,00803546 | 0,00814106 | 0,00903764 | 0,0138515 | 0,0634494 | 0,560276 |
| 32 | 0,00881882 | 0,00893341 | 0,00910083 | 0,0096234 | 0,0148982 | 0,0699609 | 0,589806 |
| 64 | 0,0114801 | 0,0120037 | 0,011589 | 0,0121289 | 0,0190697 | 0,0804543 | 0,709342 |
| 128 | 0,0583444 | 0,0557022 | 0,0500248 | 0,0545803 | 0,068489 | 0,150828 | 1,40575 |

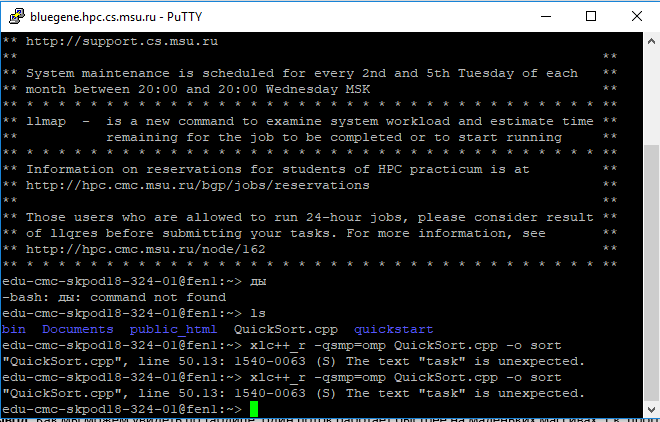
**Вывод:** Как мы можем увидеть по таблице, один поток работает быстрее на маленьких массивах, т.к. программе не требуется время, чтобы создать остальные потоки. На более больших массивах же мы получаем прирост в производительности при увеличении числа потоков, которые разбирают поочерёдно задачи, обрабатывая свои части массива, однако при чрезмерном количестве потоков программа начинает работать медленней (основное время уходит на создание потоков и распределение им определённых задач).

Оптимальным значением – является **16 потоков**, потому что они обрабатывают большие массивы быстрее, а на маленьких проигрывают десятитысячные доли секунды.

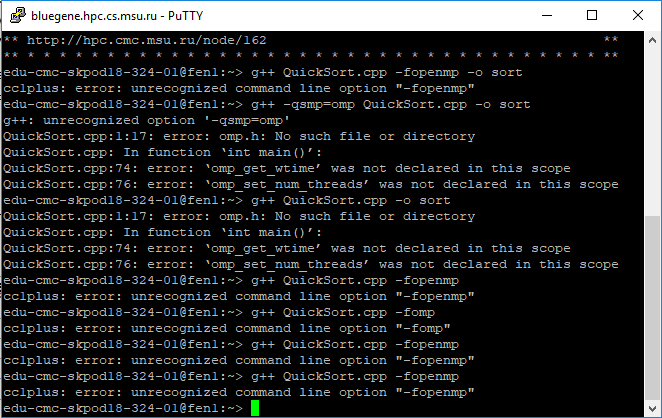
### OpenMP на Bluegene

Результаты не были получены, т.к. мне не удалось запустить компиляцию программы.

* **Компиляция программы при помощи xlc++\_r.**



* **Компиляция программы при помощи g++.**



### OpenMP на ноутбуке

* **Интересный факт** – у меня в ноутбуке стоит 4-х ядерный процессор с 8-ю потоками (по 2 потока на ядро при помощи технологии Intel Hyper-Threading). Но если изменять переменную flows на значения выше 8, программа работает и при отладке пишет, что используется поток №63 (например) и т.д.
* Это может быть связано с тем, что процессор завершает выполнение программы на одном потоке, а потом использует его же, но заменяя номер этого потока. В этом можно убедиться проанализировав диаграмму (в целом у 16 и 32 потоков одинаковые показатели, а у 64 и 128 время увеличивается из-за множественного переключения и изменения номеров потоков).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| #threads/size | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| 1 | 0.000062s | 0.000095s | 0.00062s | 0.0059s | 0.04206s | 0.4887s | 5.03891s |
| 2 | 0.000173s | 0.000220s | 0.00073s | 0.0051s | 0.04035s | 0.4524s | 4.31507s |
| 4 | 0.000241s | 0.000274s | 0.00079s | 0.0048s | 0.03908s | 0.4186s | 4.17427s |
| 8 | 0.003157s | 0.003172s | 0.00165s | 0.0113s | 0.03801s | 0.4327s | 3.92030s |
| 16 | 0.000693s | 0.000696s | 0.00158s | 0.0057s | 0.02347s | 0.3386s | 3.85402s |
| 32 | 0.000970s | 0.000479s | 0.00092s | 0.0048s | 0.02181s | 0.3912s | 3.96078s |
| 64 | 0.002371s | 0.000972s | 0.00140s | 0.0078s | 0.04597s | 0.3773s | 3.93914s |
| 128 | 0.003142s | 0.004503s | 0.00357s | 0.0037s | 0.01848s | 0.3983s | 3.90383s |

# Анализ результатов

Распараллеливание программы в среднем дало выигрыш по времени в 6,89 раза на массивах большого размера и прирост производительности на 20-30% на массивах маленького размера.

Заметим, что задача прекрасна поддалась распараллеливанию и зависимость скорости работы от числа вычислителей близка к линейной.

В сравнении с временем работы на ноутбуке принципиальный выигрыш дает выполнение на 16 вычислителях Polus. Также заметим, что на ПК уже зависимость времени работы от числа потоков не линейная, а переход от 32 потоков к 64 сопровождается спадом производительности.

# Выводы

Выполнена работа по разработке параллельной версии алгоритма быстрой сортировки массива целых чисел. Изучена технология написания параллельных алгоритмов OpenMP. Проанализировано время выполнения алгоритмов на различных вычислительных системах.

В ходе выполнения задания возникали некоторые трудности, но они постепенно решались.

Технология OpenMP крайне удобна в использовании, причем дает колоссальный прирост производительности на рассчитанных на многопоточные вычисления системах, в том числе и на персональных компьютерах.

Полный текст программы находится в файле QuickSort.cpp, также комментариями там отмечены ключевые места и некоторые важные переменные.